Compiler Directive based GPU Programming in C and Fortran

April 25, 2018

NASA Advanced Supercomputing Division

Outline

- Introduction
  - GPU Architecture
  - Concepts of GPU Programming
- OpenACC Basics
  - Off-loading Work and Data to the GPU
- Using OpenACC on Pleiades GPU Nodes
- Learning by Example
  - Sparse Conjugate Gradient Algorithm
    - Expressing Parallelism
    - Expressing Data Locality
- More OpenACC Constructs and Clauses
- What about OpenMP?
- Conclusion

Question? Use the Webex chat facility to ask the Host

CPU vs GPU

- CPU
  - Fast clock (2.4-2.9 GHz on Pleiades)
  - Multiple cores (16-40 on Pleiades)
  - Complex cores
    - Large caches, complex branch prediction, OOO execution, multi-threading
  - Parallelism
    - Deep pipelines, multiple cores, vector units
    - SIM**D** length width 16-64

- GPU
  - Slow clock (0.8-1.0 GHz)
  - Thousands of cores
    - 2880 SP cores on Pleiades
  - Light weight cores:
    - Small caches, little branch prediction, in-order execution, multi-threading
  - Parallelism: Theoretically enormous!
    - In practice limited the runtime system particular to the GPU
    - SIM**T** execution

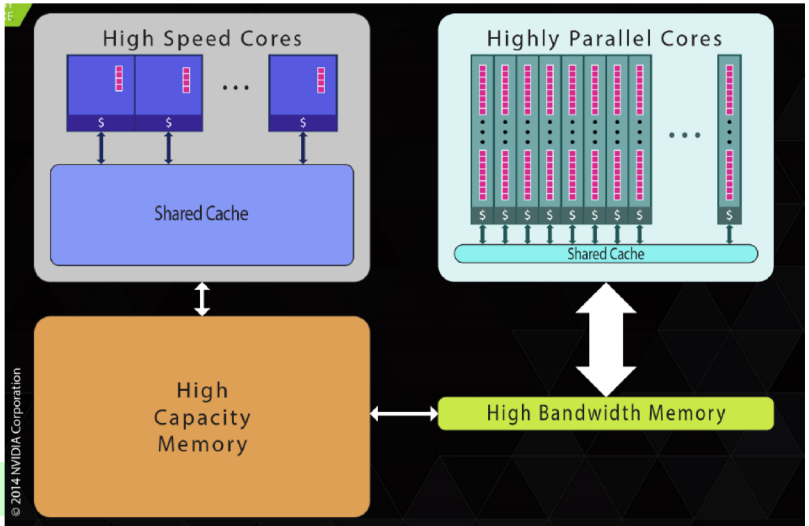
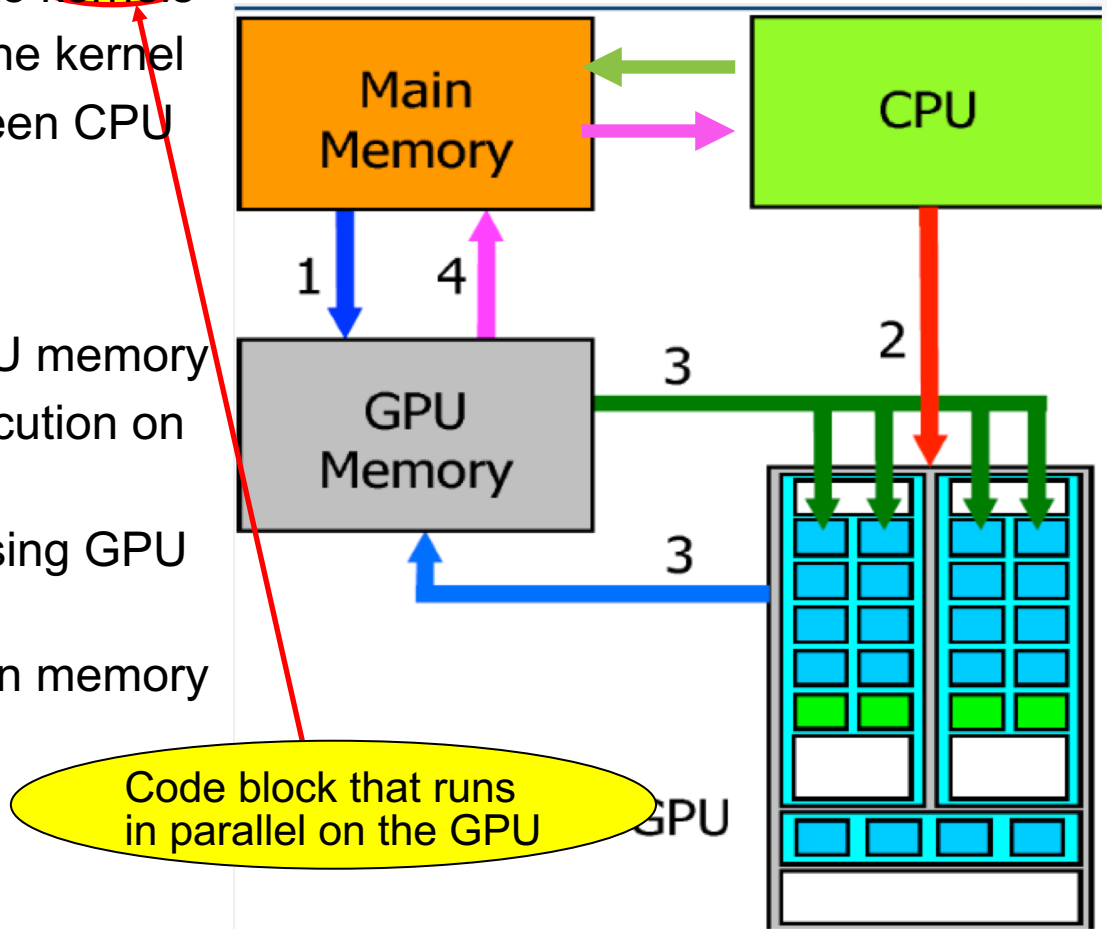


Image courtesy of Nvidia:

Flow of GPU Accelerated Computing

- GPU Accelerated Programming
  - Identify and off-load compute kernels
  - Express parallelism within the kernel
  - Manage data transfer between CPU and GPU
- Execution flow
  1. Data copy from main to GPU memory
  2. CPU initiates kernel for execution on the GPU
  3. GPU executes the kernel using GPU memory
  4. Data copy from GPU to main memory
- Programming Methods
  - Libraries
  - Directives
  - Cuda



Question? Use the Webex chat facility to ask the Host

What is OpenACC?

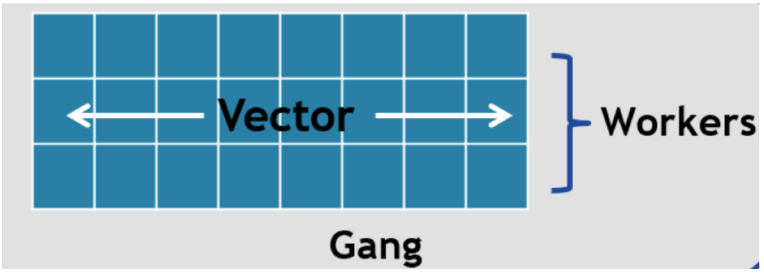
- OpenACC 2.6 Specification released 2017
  - High level parallel programming standard suitable for accelerators
- OpenACC API consists of
  - Compiler directives
  - Runtime library routines
  - Environment variables
- OpenACC provides support for
  - Identifying accelerator kernels
  - Express parallelism
  - Manage data locality
- OpenACC supports 3-level parallelism
  - Gang: 2D “block” of threads
  - Worker: Row of threads
  - Vector: Length of the row

- Example: PGI Compiler
  - OpenACC target **GPU**

Vector => thread dimension x
Worker => thread dimension y
Gang => block of threads

  - OpenACC target **CPU**

Vector => compiler’s auto-vectorization
Worker => not used
Gang => software thread
(like an OpenMP threads)



Question? Use the Webex chat facility to ask the Host

OpenACC Directive Syntax

- Compiler Directive
  - Programmer inserted hint/command for the compiler
- Directive Syntax
  - Fortran
    - Mostly paired with a matching `end directive` surrounding a structured code block

```
!$acc directive [clause [,] [clause] …]
    code
!$acc end directive
```

  - C
    - No `end directive` needed as the structured block is bracketed

```
#pragma acc directive [clause [,] [clause] …]
{
   code
}
```


Question? Use the Webex chat facility to ask the Host

OpenACC Parallel and Loop Constructs

!$acc parallel

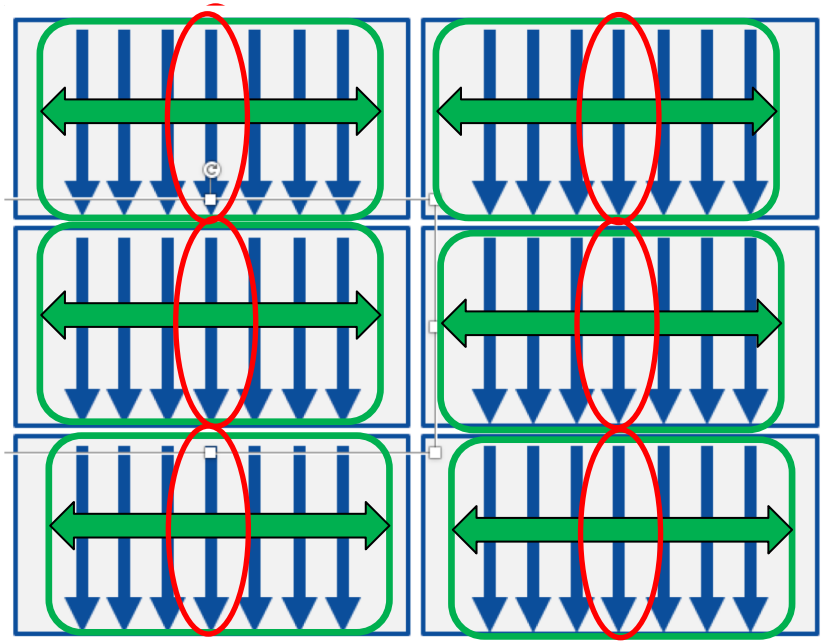
- User specifies a block of code containing parallelism
- Compiler generates GPU kernel
- Kernel is started on a set of gangs executing in parallel
- All gangs execute the same code

!$acc loop

- User identifies a loop for parallel execution
- Gangs can have multiple threads
- Compiler distributes the loop iterations across the threads

- The constructs are often placed together

```
subroutine saxpy (n, a, x, y)
   real :: x( :  ), y(: ), a
   integer :: n, I
!$acc parallel
!$acc loop
    do i = 1, n
        y(i) = y(i)+a*x( i)
    end do
```

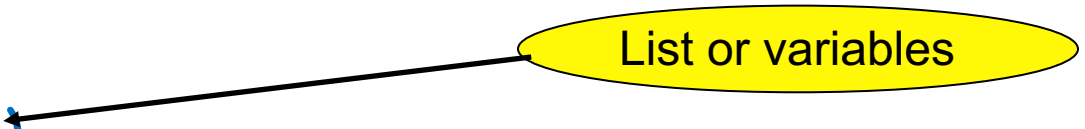


Structured Data Management: Data Construct

- By default there is no data re-use between parallel loops
  - This could lead to excessive data traffic
- Data regions:
  - A region of the program within which data is accessible to the device.
  - They can be explicitly defined to reduce data copies
  - The **`data`** construct is used to mark such regions

```
!$acc data [clauses, …] / !$acc end data
```

- Example clauses:




```
copyin (list)
```

  - Allocates memory on the GPU and copies data in when entering the region, the values are not copied back

```
copyout (list)
```

  - Allocates memory on GPU and copies the data to the host when exiting the region

```
present (list)
```

  - The data is already present on the GPU

Question? Use the Webex chat facility to ask the Host

Unstructured Data Management: Enter/Exit Data Constructs

- Real life applications are not always a nicely structured sequence of loops
  - Subroutine calls
  - C++ Structures or Fortran user defined types with dynamic arrays
- Unstructured data directives

```
!$acc enter data
```

  - Allocate memory on the device for the remainder of the program or until explicitly deleted
  - Possible clauses are `copyin` and `create`

```
!$acc exit data
```

  - Deallocate the memory on the device
  - Possible clauses are `copyout` or `delete`
- Multiple enter/exit data constructs, branched across different function calls are allowed

Question? Use the Webex chat facility to ask the Host

Compiling and Running on Pleiades

- Compilation

```
module load comp-pgi cuda
pgf90 -o cg.x –fast -ta=tesla,lineinfo -Minfo=acc cg.f90
pgcc -o cg.x -fast -ta=tesla,lineinfo -Minfo=acc cg.cpp
```

load the PGI compiler and CUDA

- Submit to GPU node

```
qsub -l select=1:ncpus=16:model=san_gpu -q k40
```

- Load the same modules as for compilation
- Optional: To obtain timing information set

```
setenv PGI_ACC_TIME
```

- Run the executable

```
./cg.x
```

- For more detailed performance information use the GPU profiler

```
nvprof/pgprof ./cg.x
```

Get Information about your GPU Card

PGI Tool pgaccelinfo

Pleiades GPU Card

```
Device Number:                  0
Device Name:                    Tesla K40m
Device Revision Number:         3.5
Global Memory Size:             11995578368
Number of Multiprocessors:      15
Number of SP Cores:             2880
Number of DP Cores:             960
Concurrent Copy and Execution:  Yes
Total Constant Memory:          65536
Total Shared Memory per Block:  49152
Registers per Block:            65536
Warp Size:                      32
Maximum Threads per Block:      1024
Maximum Block Dimensions:       1024, 1024, 64
Maximum Grid Dimensions:        2147483647 x 65535 x 65535
Maximum Memory Pitch:           2147483647B
Texture Alignment:              512B
Clock Rate:                     745 MHz
Execution Timeout:              No
Integrated Device:              No
Can Map Host Memory:            Yes
Compute Mode:                   default
Concurrent Kernels:             Yes
ECC Enabled:                    Yes
Memory Clock Rate:              3004 MHz
Memory Bus Width:               384 bits
L2 Cache Size:                  1572864 bytes
Max Threads Per SMP:            2048
Async Engines:                  2
Unified Addressing:             Yes
Managed Memory:                 Yes
PGI Compiler Option:            -ta=tesla:cc35
```

Example: Sparse Conjugate Gradient

Basic Structure

```
do while (norm .ge. tol)
   …
   call dot (r, r)
    …
   call waxpby (one, r, beta, p, p)
   …
   call matvec (a, p, ap)
   …
   call waxpby (one, x, alpha, p, x)
   call waxpby (one, r, -alpha, ap, r)
   …
enddo
```

We focus on sparse matrix-vector multiply

F90 Sparse Matvec Basic…

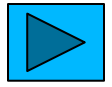
```
do while
  …
!$acc parallel loop
do i=1,a%num_rows
   tmpsum = 0.0d0
   row_start = arow_offsets(i)
   row_end   = arow_offsets(i+1)-1
!$acc loop
   do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum +
             acoef*xcoef
   enddo
   y(i) = tmpsum
enddo
…
enddo
```

pgf90 -fast -ta=tesla:cc35, lineinfo -Minfo=acc

```
matvec:
     Accelerator kernel generated
     Generating Tesla code
 115, !$acc loop gang
 120, !$acc loop vector(128)
      Generating implicit reduction(+:tmpsum)
 114, Generating implicit
copyin(acols(:),arow_offsets(1:a%num_rows+1),acoefs(:))
      Generating implicit copyout(y(:a%num_rows)
      Generating implicit copyin(x(:))
```

Reported Timings:

Total Iterations: 100 **Time (s): 148.7404**



Question? Use the Webex chat facility to ask the Host

F90 Sparse Matvec Better …

```
!$acc enter data copyin (x, a, a%row_offsets,a%cols,a%coefs)
!$acc enter data create (y)
```

type/structure

After allocation and initialization on the host
Before the while loop
Can be in a different function call

```
do while (norm .gt. tol) {
  …
!$acc parallel loop
!$acc& present(x,y,arow_offsets,acols,acoefs)
  do i=1,a%num_rows
   tmpsum = 0.0d0
   row_start = arow_offsets(i)
   row_end   = arow_offsets(i+1)-1
!$acc loop reduction(+:tmpsum)
   do j=row_start,row_end
     acol = acols(j)
     acoef = acoefs(j)
     xcoef = x(acol)
     tmpsum = tmpsum +
              acoef*xcoef
    enddo
    y(i) = tmpsum
    enddo
   …
 enddo
```

```
Generating
present(arow_offsets(:),y(:),x(:),acols(:),acoefs(:))
    Accelerator kernel generated
    Generating Tesla code
120,    !$acc loop gang    ! blockidx%x
125,    !$acc loop vector(128)    ! threadidx%x
        Generating reduction(+:tmpsum)
```

Reported Timings:
Total Iterations: 100 **Time (s):** 16.22556

Question? Use the Webex chat facility to ask the Host

F90 Sparse Matvec Best!

```
!$acc enter data copyin (a,a%row_offsets,a%cols,a%coefs)
!$acc enter data create(y)
```

```
do while
  ...
!$acc parallel loop &
!$acc& present(x,y,arow_offsets,acols,acoefs)&
!$acc& gang worker num_workers(32) vector_length(32)
 do i=1,a%num_rows
   tmpsum = 0.0d0
   row_start = arow_offsets(i)
   row_end   = arow_offsets(i+1)-1
!$acc loop reduction(+:tmpsum)vector
   do j=row_start,row_end
     acol = acols(j)
     acoef = acoefs(j)
     xcoef = x(acol)
     tmpsum = tmpsum +
              acoef*xcoef
    enddo
    y(i) = tmpsum
   enddo
   ..
enddo
```

Inner loop is only 27 iterations
Better to exploit parallelism
across workers and and vectors

```
Generating present(arow_offsets(:),y(:),x(:),acols(:),acoefs(:))
    Accelerator kernel generated
    Generating Tesla code
120,    !$acc loop gang, worker(32)    !blockidx%x threadidx%y
125,    !$acc loop vector(32)      !threadidx%x
        Generating reduction(+:tmpsum)
```

Reported Timings:
Total Iterations: 100 **Time (s): 5.863531**

PGI_ACC_TIME Output

```
matrix.F90  matvec  NVIDIA  devicenum=0    time(us): 26,910,583
114: compute region reached 101 times
114: kernel launched 101 times
     grid: [65535]  block: [128]
     elapsed time(us): total=15,879,014
     max=157,383 min=156,798 avg=157,217
114: data region reached 202 times
114: data copyin transfers: 17069
      device time(us): total=26,284,387 max=1,686 min=4 avg=1,539
129: data copyout transfers: 404 device time(us): total=626,196
     max=1,60  min=1,396 avg=1,549
```

MATVEC Basic

```
matvec  NVIDIA  devicenum=0
   time(us): 5,385,152
   118: compute region reached 101 times
        118: kernel launched 101 times
            grid: [65535]  block: [32x32]
             device time(us): total=5,385,152 max=53,340 min=53,295 avg=53,318
            elapsed time(us): total=5,387,929 max=53,369 min=53,322 avg=53,345
   118: data region reached 202 times
```

MATVEC Best

Question? Use the Webex chat facility to ask the Host

C++ Sparse Matvec Best

```
#pragma acc enter data copyin(A)
#pragma acc enter data \
copyin(A.row_offsets[:num_rows+1],A.cols[:nnz],A.coefs[:nnz])
```

```
…
{
#pragma acc parallel loop present (row_offset,cols,Acoefs,xcoefs, ycoefs)\
gang worker num_workers(32) vector_length(32)
   for(int i=0;i<num_rows;i++) {
     double sum=0;
     int row_start=row_offsets[i];
     int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
     for(int j=row_start;j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
      }
      ycoefs[i]=sum;
    }
} while (norm > tol)
```

Reported Timings:
Total Iterations: 100 **Time (s): 4.908**

```
Generating present(arow_offsets(:),y(:),x(:),acols(:),acoefs(:))
    Accelerator kernel generated
    Generating Tesla code
120,     !$acc loop gang, worker(32)
125,     !$acc loop vector(32)
         Generating reduction(+:tmpsum)
```

Using CUDA Unified (Managed) Memory

```
!$acc parallel loop
do i=1,a%num_rows
 tmpsum = 0.0d0
 row_start = arow_offsets(i)
 row_end   = arow_offsets(i+1)-1
!$acc loop
 do j=row_start,row_end
   acol = acols(j)
   acoef = acoefs(j)
   xcoef = x(acol)
   tmpsum = tmpsum + acoef*xcoef
 enddo
 y(i) = tmpsum
enddo
```

- Technology that allows a single pointer to be dereferenced either CPU or GPU
- The CUDA driver will migrate pages if required
- The PGI supports using this feature

- Dynamically allocated data in managed/unified memory
- Requires the PGI compiler,
- Might not always be profitable
- Might run into system errors

pgf90 -fast -ta=tesla:cc35, **managed**, lineinfo -Minfo=all

Reported Timings:
Total Iterations: 100 **Time (s):** 16.22556

Question? Use the Webex chat facility to ask the Host

The Kernels Construct

- `#pragma acc kernels`
  - Compiler generates accelerator kernel(s) for the code region
  - Parallelism is based on compiler dependence analysis
  - Pro:
    - The user is not responsible for expressing parallelism
    - The `loop` directive is not necessary to distribute the work
    - Code region may contain multiple loops
    - Fortran array syntax can be correctly parallelized
  - Con:
    - Compiler might not be able to detect parallelism e.g you might want to add the "`restrict`" keyword or the compiler flag "`-Msafeptr`"

```
subroutine saxpy (n, a, x, y)
   real :: x( :  ), y(: ), a
   integer :: n, I
!$acc acc kernels
    do i = 1, n
       y(i) = 1.0
       x(i) = 2.0
    end do
    do i = 1, n
        y(i) = a * x( i) + y(i)
    end do
!$acc end kernels
```


Question? Use the Webex chat facility to ask the Host

Some Golden Rules for Optimization

- Optimizing memory traffic:
  - Use the PGI compiler with the –ta=managed during development
  - Use a profiler or the compiler to analyze data movement
  - Add data directives to mimic the behavior
- For Nvidia GPUs the **vector length to be a multiple of 32**
  - Ensures full use of *warps*
  - *Warps* are groups of 32 GPU SIMT threads
- In general pick **vector_length * num_workers >= 128**
  - Try hitting 128, 256, 512, 768, and 1024 and pick which is the best
- Good approach for the beginner: Use the ***kernels*** construct first, then optimize using the ***parallel loop*** construct

Performance Challenges

- Function calls in inner loops:
  - Challenging for the compiler's dependence analysis
    - Possibly use `$!acc routine` compile routine for the device or in-line
- IF-Branches:
  - Can get expensive if threads take different execution paths
    - Move your branches up to a level in your code where all threads go down the same code branch
    - Avoid branches in inner loops
- Structures with complex and/or dynamic components
- Not much synchronization support
  - Restructure code so that no synchronization is necessary
- Loop strides/memory layout
  - Good memory access pattern is essential
  - Inner loop should move along the fastest dimension
  - Fastest dimension should be long

Question? Use the Webex chat facility to ask the Host

What about Using OpenMP?

- OpenMP 4 provides constructs to off-load code to a device manage the data
- OpenMP did follow the example of OpenACC, but is more inclusive
- Important constructs and clauses:

  `#pragma omp target` or `$!omp target`

  - Create data environment and execute code region on the device

  `#pragma omp target map(map-type: list)`

  - Map a variable to/from the device data environment

  `#pragma teams distribute, parallel for, simd`

  - Distribute the work

Laplace Solver in **OpenACC**

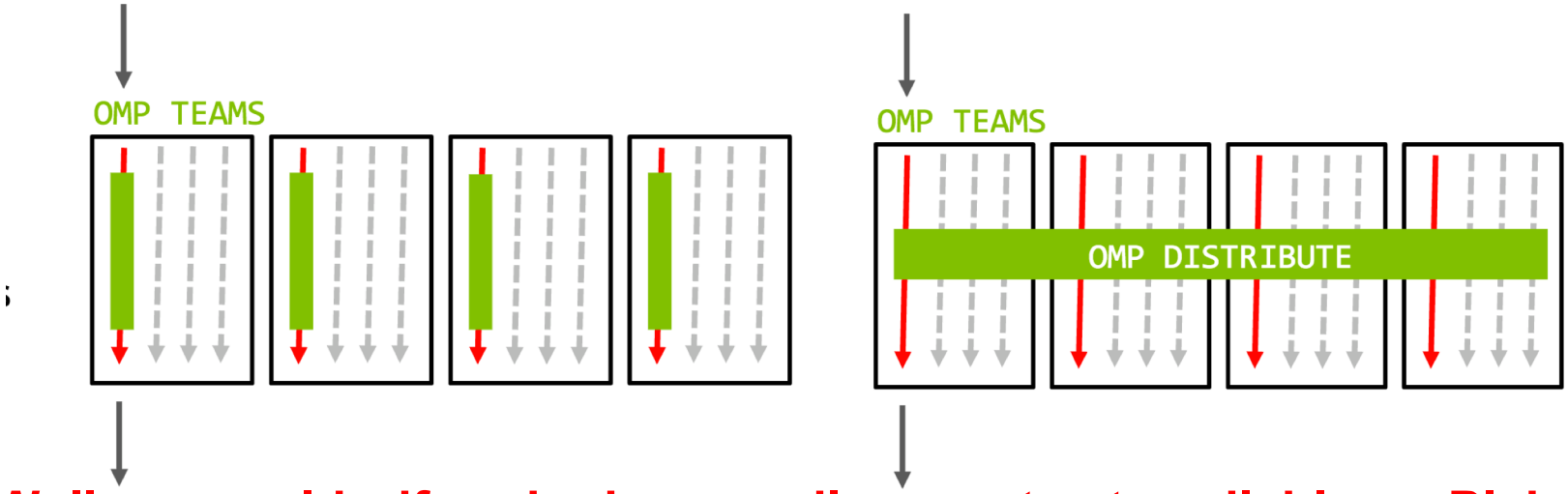
```
#pragma acc parallel loop
   for( int j = 1; j < n-1; j++{
#pragma acc loop reduction(max:error)
        for( int i = 1; i < m-1; i++ )                {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
           }        }
```

Can we do this in OpenMP?

Question? Use the Webex chat facility to ask the Host

Yes, We Can!

```
{
#pragma omp target teams distribute
   for( int j = 1; j < n-1; j++)   {
#pragma parallel for reduction(max:error)
        for( int i = 1; i < m-1; i++ )              {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
           }        }
```



Well, we could…if we had a compiler… not yet available on Pleiades

For more details check out the presentation by Jeff Larkin, Nvidia:
http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf

Image courtesy of Nvidia

Question? Use the Webex chat facility to ask the Host

Summary

- Compiler directives provide a quick path to run codes on the GPU
- Pro's:
  - Compiler directives allow single source code
    - No need to maintain multiple code paths
  - High level
    - Abstract away device details, focus on expressing parallelism and data locality
  - Optimizations for device specifics left to compiler and device driver
- Con's:
  - Dependence on compiler quality
    - OpenACC stable support by PGI, Cray, GNU gcc
    - OpenMP support by Cray, GNU gcc, clang
- Benefit of effort to port to a GPU:
  - Restructuring the code for the GPU often yields considerable performance increase for the CPU

References

Using GPUs codes on Pleiades HECC KB

https://www.nas.nasa.gov/hecc/support/kb/using-gpu-nodes_298.html

OpenACC Home Page and Books

https://www,openacc.org

https://www.amazon.com/OpenACC-Programmers-Strategies-Sunita-Chandra
https://www.amazon.com/Parallel-Programming-OpenACC-Rob-Farber/dp/0124103979

https://www.amazon.com/Programming-Massively-Parallel-Processors-Hands/dp/0128119861

Jeff Larkin AMS Presentation

https://www.nas.nasa.gov/publications/ams/2015/04-21-15.html

- OpenMP Home Page and Books

http://www.openmp.org/

http://www.openmp.org/tech/using-openmp-next-step/

https://mitpress.mit.edu/books/using-openmp

- PGI Compiler HECC KB

https://www.nas.nasa.gov/hecc/support/kb/PGI-Compilers-and-Tools_365.html

Question? Use the Webex chat facility to ask the Host

Q&A

How can I run MPI + OpenACC codes on Pleiades?

This is described on slide 28 in the backup material

How does the K40 compare to other GPUs?

The new P100 and V100 GPUs are 2-4X faster than the K40 so one should not make a final performance judgment using the K40s

Why do you have to copy the whole data structure a and its elements a%*?

" deep data copies" are currently not supported by the PGI compiler. The OpenACC standard did add manual deep copy (attach/detach) to the 2.6 Spec. A current proposal is here: https://www.openacc.org/sites/default/files/inline-files/TR-16-1.pdf. In Fortran, there is a deep-copy flag (-ta=tesla:deepcopy) that you can try. It's a PGI extension and not part of OpenACC. OpenMP 5.0 may include support for deep copies of pointer chasing structures via user defined mappers, which can be used in the data clause.

Can I have calls to math intrinsic functions (eg sin, cos) in an OpenACC parallel region?

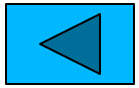
Basic routines like sin, cos, exp, which are part of the language are on the GPU and work with OpenACC. More discussion on calls within parallel regions is on slide 21.

PGI does not support OpenMP target off-load. How can I add off-load to my existing OpenMP code?

You could use the PGI compiler and add the "-mp" flag in addition to –ta=tesla:cc35. This will enable both directives.

```
type matrix
 sequence
 integer :: num_rows
 integer :: nnz
 integer, pointer ::
   row_offsets(:)
 integer, pointer :: cols(:)
 real(8), pointer :: coefs(:)
end type matrix
```

```
  struct matrix {
  unsigned int num_rows;
  unsigned int nnz;
  unsigned int *row_offsets;
  unsigned int *cols;
  double *coefs;
};
```



Multi-node runs on Pleiades

- Compilation

```
module load comp-pgi cuda mpi-hpe
mpif90 -o cg.x –fast -ta=tesla,lineinfo -Minfo=acc cg.f90
setenv MPICC_CC pgcc
mpicc -o cg.x –fast -ta=tesla,lineinfo -Minfo=acc cg.cpp
```

- Submit to GPU Nodes

```
qsub -l select=4:ncpus=16:model=san_gpu -q k40
```

- Load the same modules as for compilation
- Set environment variable to tell MPI that CUDA is being used

```
setenv MPI_USE_CUDA 1
 setenv MPI_SHEPHERD 1
```

- Run the executable

```
mpiexec –np 4 ./cg.x
```